Control States for Atlas Framework

Paolo Calafiura, LBL

Jim Kowalkowski, Charles Leggett, John Milford,
Marjorie Shapiro, Craig Tull, Laurent Vacavant

Summary

- ➢ Control Framework: What is it?
- ➢ Lassi’s Object Networks
- ➢ What we want to add to them
- ➢ Design: System Features
- ➢ Design: The core classes
- ➢ Design Scenarios
- ➢ Status

What Is It?

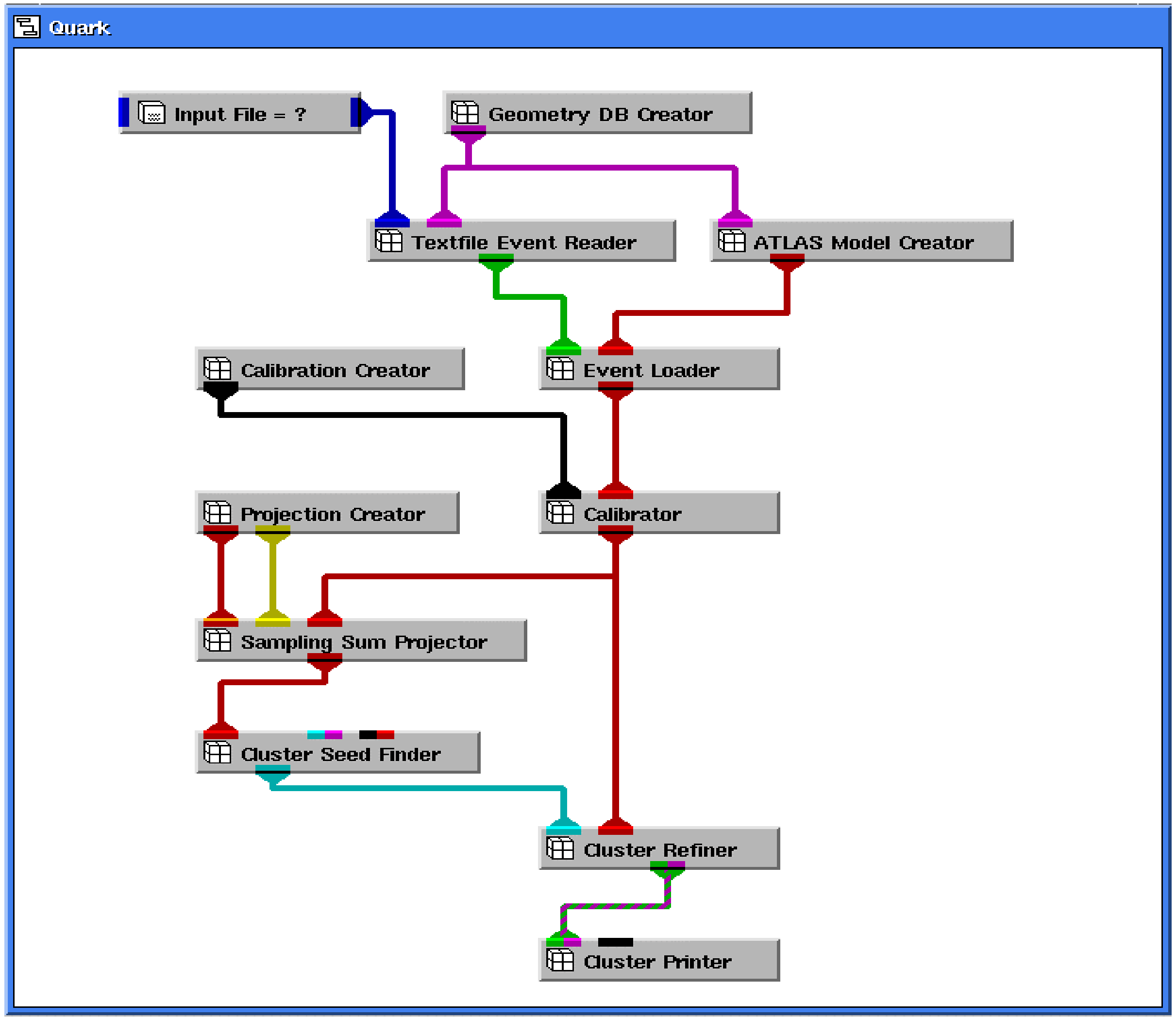
The control framework is the part of the infrastructure that makes sure that

- The right piece of software
- Runs
- At the right time
- With the right inputs and
- The outputs go to the right place

(Lassi's definition)

Lassi's Object Networks

- ➢ Colors = data types
- ➢ Modules = behavior
- ➢ Whole network = component
- ➢ Input-output dependency



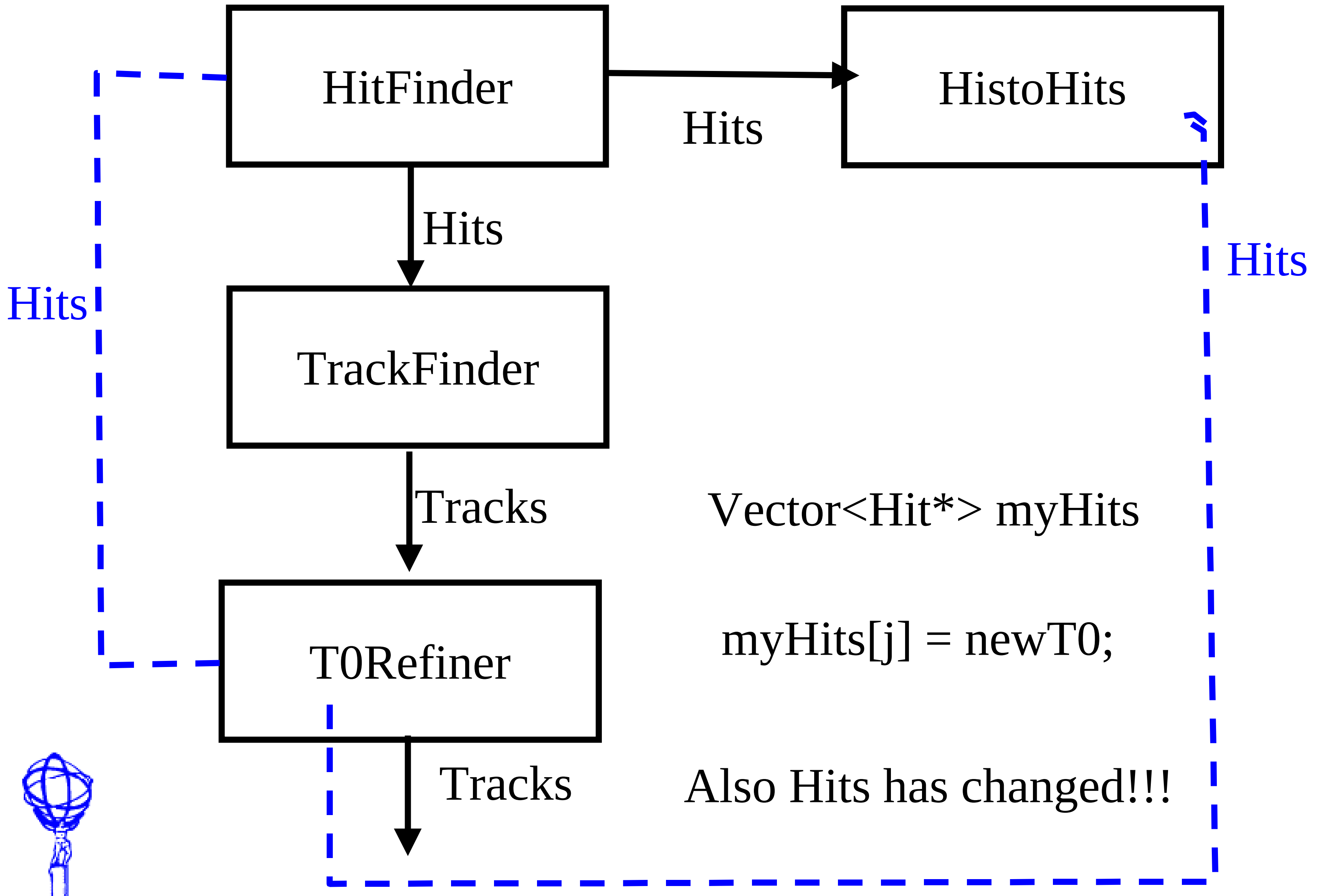
Object Networks Features

✓ Design based on *components*
- Implementing well-defined *interfaces*
- Extensive use of *notification*
- Goal is to *maximize re-usability*

✓ Data flow based, pushing data down to trigger execution
- Indeed like a trigger system
- Kind of natural way to design a reconstruction program

➢ Is this the way we **think** when we **analyze** the data?
- No! We **pull** data at random (well…) from the modules that reconstructed them, after they are done for that event (run, job,…)

➢ How easy will be to predict (and repeat!) the execution path of a 1000 objects network?



Vector<Hit*> myHits

myHits[j] = newT0;

Also Hits has changed!!!

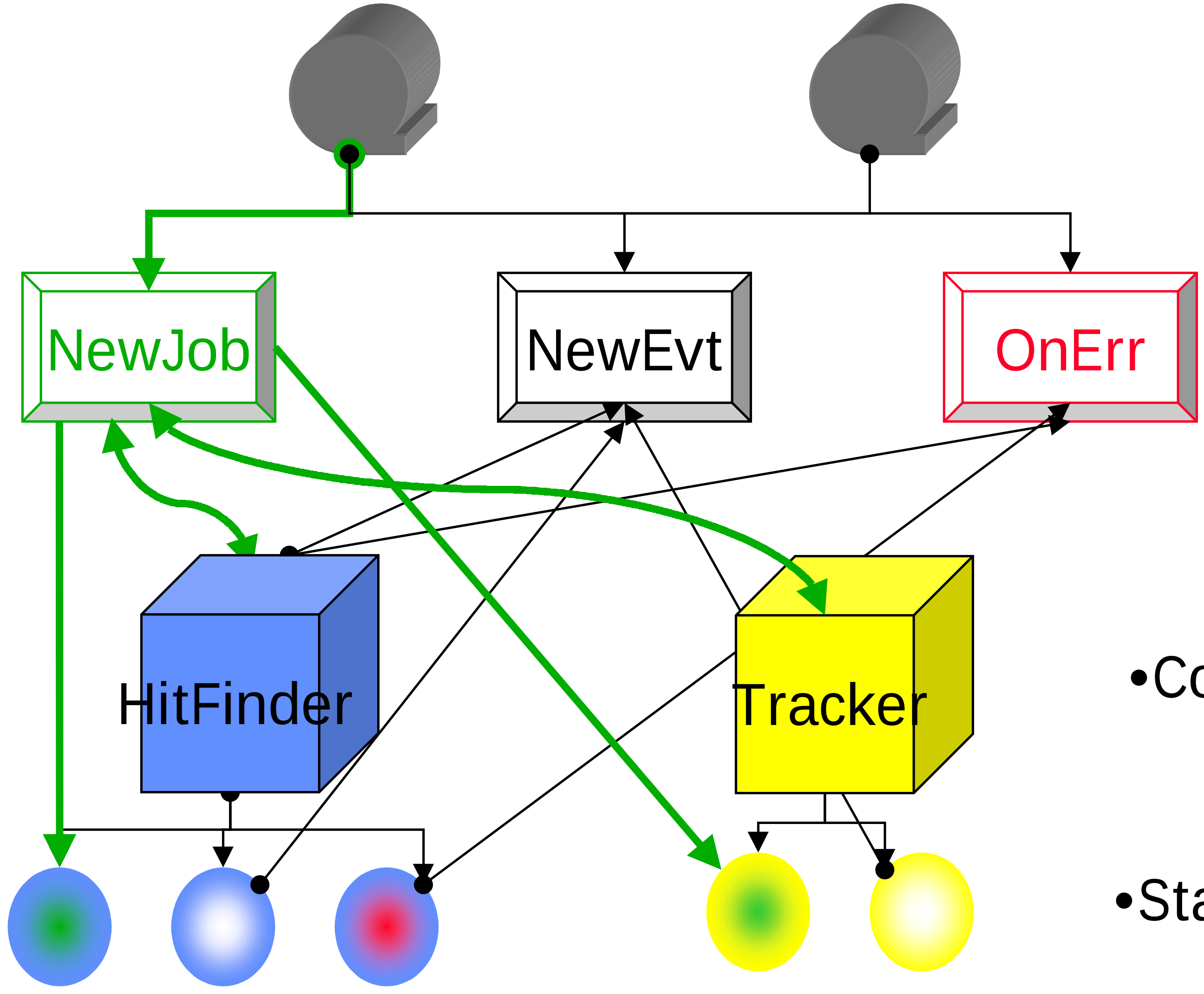
What Is Missing?

- ➢ I don't think we can reasonably interact with a self-triggering network of say 1000 components without knowing its global **state.**
- ➢ The framework as a State Machine:
  - My HFILL must run after "event done"
  - My new geometry constants must be loaded for "run 4567"
  - I have to broadcast a "pack-up and go" message to 1000 modules when the muon decoding module produces a "fatal error"

Solution: Add Control States to the Network

- Synchronize network execution, notifying modules about the next state transitions they may be interested into
- Control (or, even better, to suggest) the order in which the components undergo a state transition (=run)
- Define the states, the order of modules and the state sources, dynamically via the UI
- There should be no linker dependencies among components and framework

The Control States Network



- •Sources
- •States
- •Components
- •State Methods

Setting up - a sample script

- associate States and StateSources

```
StateSource rawFile(inputFile)
next_event.attach(rawFile)
```

- define Sequences of components to be executed

```
sequence all =
        { "hitFinder", "tracker", "myanal" }
sequence reco = { "tracker", "myanal"}
```

- define State transitions, with usual flow-control constructs

```
next_run.run("all")
while (next_event.run("all")) {
          fill_histos.run("reco")
          fill_Bhistos.run("paolo")
}
```

The Component Interface Dictionary

- describe to the framework (via code generation)
  - the States

```
interface next_event : State {};
```

  - each component interface

```
interface hitFinder {
  void init();
  Result nextEvent(in WireCollection wires,
                   out HitsCollection hits);
}
```

  - the association between States and component methods

```
ADD_STATE_METHOD(next_event,
                 hitFinder::nextEvent)
```

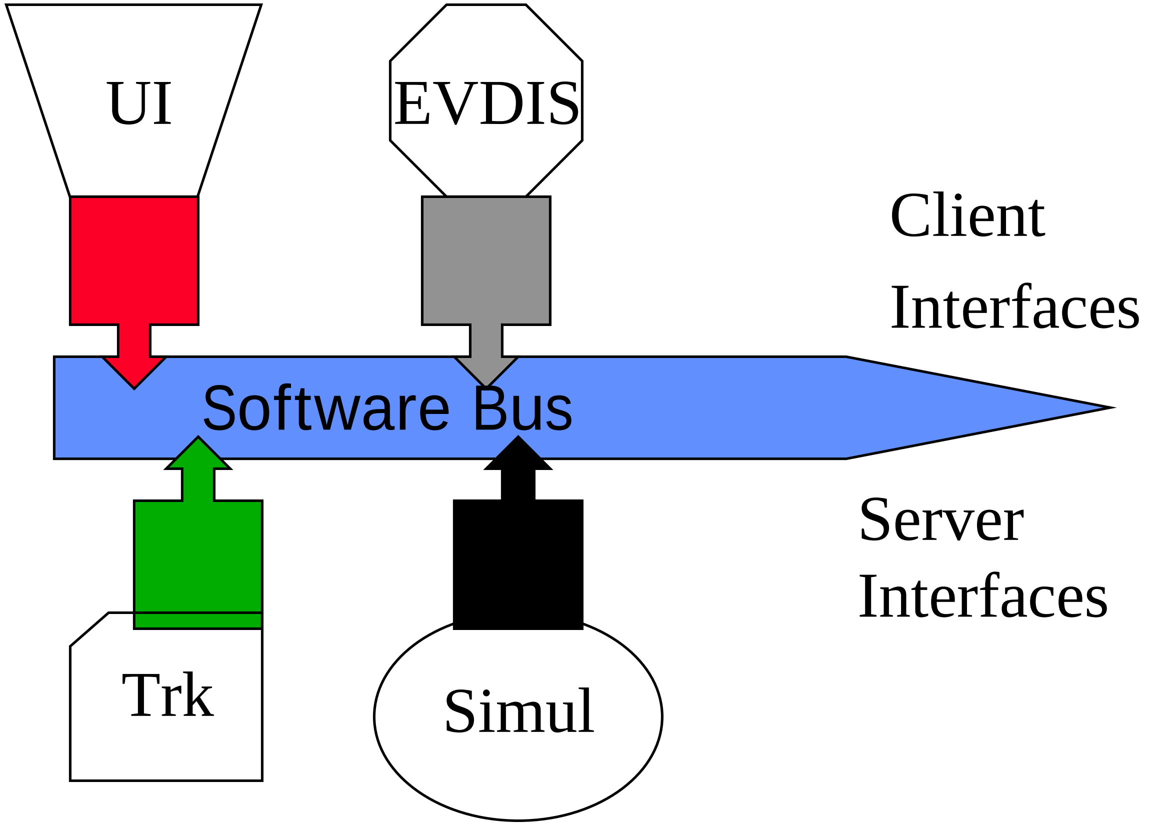
Running

- The framework runs States following the script order.
- Control returns to the framework after each state completes
- The State tries to run each registered component in order
- The Component determines what is the status of its associated method (e.g Ready, notReady, alreadyRun), run it if ready, and report to the State.
- The Object Network (or a Data Manager) notifies Components when their Parameters are ready or change.
- The State may re-queue a Component which is NotReady.

The Core Classes

- State Source
  - drive the framework generating actions
- State (and Concrete States)
  - observe sources for matching actions, run component methods
- Component Managers
  - observe states, add matching methods to their queue
  - generated from dictionary
- Component Methods
  - implement the software-bus concept
  - function objects wrapping real component method
  - determine their status
  - marshal parameters (database, F77)
  - generated from dictionary

The Software Bus



A Toy Implementation

```
class HitFinder__newEvent : public virtual IRunnable {
      .....
    //IRunnable implementation
    inline IRunnable* clone() const { return new __newEvent(*this);}
    Result run(const IScheduler& s) {
      Result rc;
      Handle < TrackSet >  set1;
      Handle < TrackSet >  set2;
      Container < ParticleSet >  set3;
      Key < TrackSet >  key1("COT");
      Key < TrackSet >  key2("SVX");
      Key < ParticleSet >  key3("chargedCandidates");
      //unlikely to be done exactly like that but...
      event->get(key1, set1);
      event->get(key2, set2);
      rc = _comp->newEvent(set1, set2, set3);
      if (rc == Result::success)
              event->put(key3, set3);
    }
};
```

Scenario: Running a State

- The source notifies all registered states that he has a newEvent action

```
StateSource::notify DEBUG: notifying newEvent
```

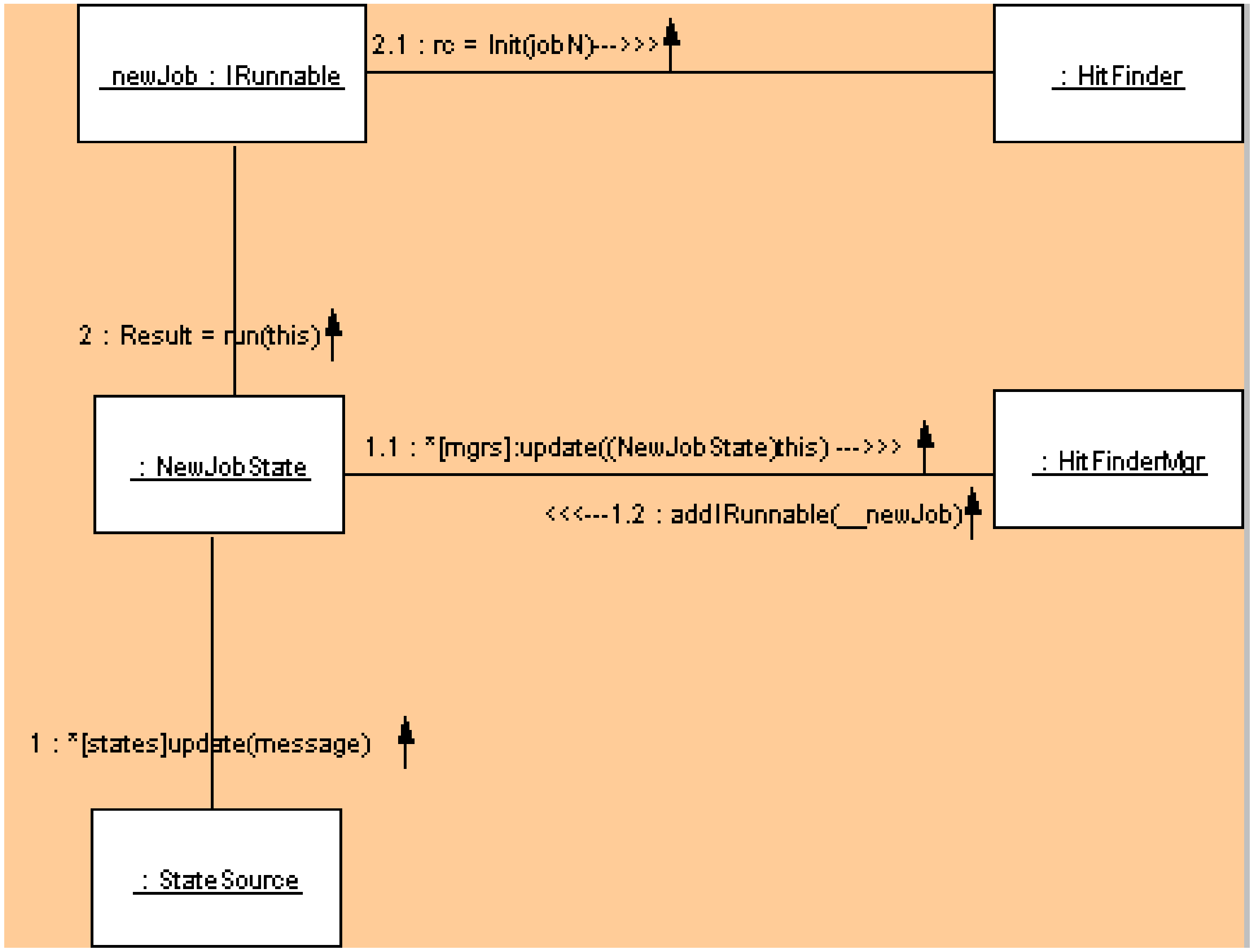
- newEvent state catches the action and notifies its observers, the managers

```
State::update DEBUG: newEvent[instanceof NewEventState] got
message newEvent
```

- Each manager add the matching method to the state queue
- Now newEvent runs the scheduled methods

```
State::run DEBUG: newEvent[instanceof NewEventState] starts
Hitfinder::newEvent DEBUG: running
State::run WARNING: newEvent[instanceofHitFinder::__newEvent]
was not ready and had to be rescheduled
Histogrammer::newEvent DEBUG: running
Hitfinder::newEvent DEBUG: running
```

Scenario: Running a State


newJob : IRunnable
2.1 : rc = Init(jobN)--->>>
: HitFinder
2 : Result = run(this)
: NewJobState
1.1 : *[mgrs]:update((NewJobState)this) --->>>
: HitFinderMgr
<<<---1.2 : addIRunnable(__newJob)
1 : *[states]update(message)
: StateSource

Scenario: Setting Up

- First we define the state classes

```
DEFINE_CTRL_STATE(NewJobStateS)
DEFINE_CTRL_STATE(NewRunState)
DEFINE_CTRL_STATE(NewEventState)
```

- Then we create the component managers

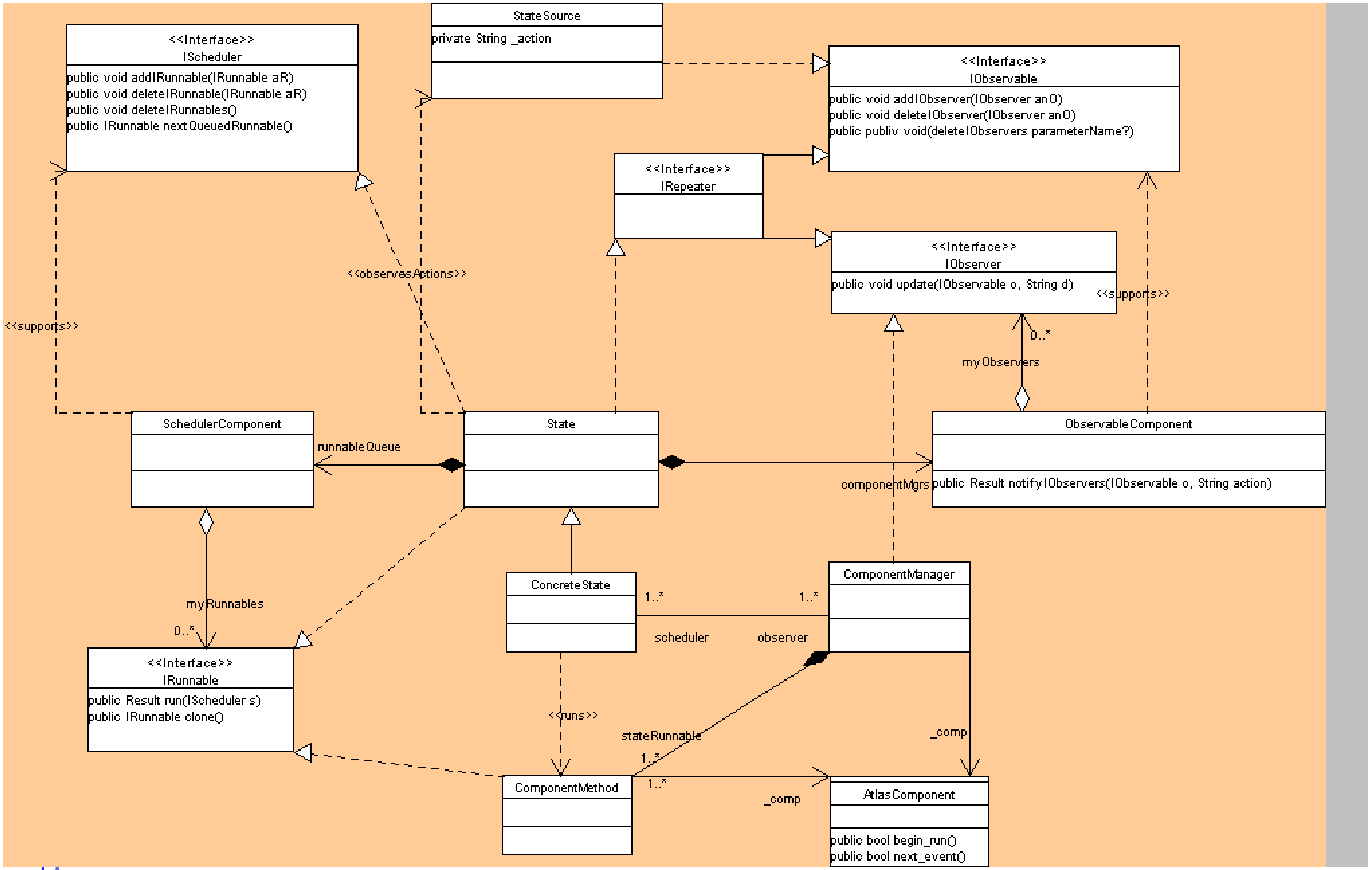
```
HitFinderMgr hitFinder;
HistogrammerMgr myHistos;
```

- We create the states instances and we register the component with them.

```
NewJobState newJob("newJob");
newJob.addIObserver(&myHistos);
newJob.addIObserver(&hitFinder);
```

- Finally we create the state source and register the states with it.

```
StateSource testSource("testSource");
testSource.addIObserver(&newJob);
testSource.addIObserver(&newRun);
testSource.addIObserver(&newEvent);
```


StateSource
private String _action
<<Interface>>
IScheduler
public void addIRunnable(IRunnable aR)
public void deleteIRunnable(IRunnable aR)
public void deleteIRunnables()
public IRunnable nextQueuedRunnable()
<<Interface>>
IObservable
public void addIObserver(IObserver anO)
public void deleteIObserver(IObserver anO)
public publiv void(deleteIObservers parameterName?)
<<Interface>>
IRepeater
<<Interface>>
IObserver
public void update(IObservable o, String d)
<<observesActions>>
<<supports>>
<<supports>>
0..*
myObservers
SchedulerComponent
runnableQueue
State
ObservableComponent
componentMgrs
public Result notifyIObservers(IObservable o, String action)
myRunnables
0..*
ConcreteState
ComponentManager
1..*
1..*
scheduler
observer
<<Interface>>
IRunnable
public Result run(IScheduler s)
public IRunnable clone()
<<runs>>
stateRunnable
_comp
1..*
1..*
ComponentMethod
_comp
AtlasComponent
public bool begin_run()
public bool next_event()

Where do we stand?

- ➢ We have a web page
  `http://iago.lbl.gov/paolo/ATLAS/framework/actiondesign.html`
- ➢ We have a prototype (can get it from the same URL)
  - Core classes running
  - Interface dictionary starting
  - Scripting in progress (IDL to Swig, John M.)
- ➢ We can use the prototype as a test bed for the requirements and use-cases exercises in progress